**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All input from untrusted sources must be validated, including but not limited to user inputs, command line arguments, network data, environment variables, and files. Input validation that is properly followed can help secure against the majority of software vulnerabilities. |
| 1. Heed Compiler Warnings | Ensure that any code compiled is done so with the highest warning level available for the compiler, as well as ensuring that any warnings are immediately addressed. Utilize various static and dynamic tools to help detect and eliminate any additional security vulnerabilities. |
| 1. Architect and Design for Security Policies | Develop software architecture that implements and enforces security policies. If a system requires varying levels of privileges at different intervals, then implement a separation of concern amongst the distinct intercommunication subsystems with appropriate privileges. |
| 1. Keep It Simple | Design software to be as simple and small as possible. Software that is overly complex introduces more errors. Also, as a system increases in complexity so does the requirement for more complex security mechanisms. |
| 1. Default Deny | Always base access upon permission rather than exclusion, effectively ensuring that access is denied initially with the protection scheme inherently identifying the necessary conditions for permission. |
| 1. Adhere to the Principle of Least Privilege | Every process implicit in a program should operate on the principle of least privilege. Ensuring that only necessary privileges are provided to complete a job. Also, any enhanced permissions are allowed for the least amount of time possible. This effectively reduces the possibility of an attacker executing arbitrary code to gain unauthorized access. |
| 1. Sanitize Data Sent to Other Systems | Ensure all data sent to complex subsystems is sanitized, including but not limited to command shells, relational databases, and any commercial off-the-shelf (COTS) components. Attackers will be able to access unused functionality in these components by way of SQL, command, or other injection methods. |
| 1. Practice Defense in Depth | Manage risk by utilizing multiple defense strategies, ensuring that if one layer of dense is compromised, additional layers can successfully prevent any flaws from becoming exploitable vulnerabilities. The use of secure programming techniques with secure environments can reduce the likelihood that vulnerabilities remaining in code at deployment are exploitable. |
| 1. Use Effective Quality Assurance Techniques | Robust quality assurance techniques are effective at identifying and eliminating vulnerabilities. Techniques such as fuzz testing, penetration testing, and source code audits should all be required to be incorporated when employing effective quality assurance. Independent security reviews lead to more secure systems, while external reviewers help bring greater perspectives in identifying and correcting invalid assumptions. |
| 1. Adopt a Secure Coding Standard | Secure coding standards are to be included in any and all target development languages and platforms. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | Do not define a C-style variadic function |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | C-style variadic functions are considered unsafe as they lack type checking and allow for mismatched arguments, resulting in easily exploitable undefined behavior. The DCL50-CPP standard recommends avoiding these functions and using function parameter in situations in which “a variable number of arguments should be passed to a function. (*DCL50-CPP. Do Not Define a C-style Variadic Function - SEI CERT C++ Coding Standard - Confluence*, n.d.)” Also, function currying should be used in place of variadic functions.  https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL50-CPP.+Do+not+define+a+C-style+variadic+function |

| **Noncompliant Code** |
| --- |
| The non-compliant code block utilizes the add() function that accepts a variable number of integer arguments. If a non-int argument or incorrect number of arguments are passed, the function will result in undefined behavior. This code block represents potential vulnerabilities when utilizing non-compliant secure coding methods. |
| #include <cstdarg>    int add(int first, int second, ...) {  int r = first + second;  va\_list va;  va\_start(va, second);  while (int v = va\_arg(va, int)) {  r += v;  }  va\_end(va);  return r;  } |

| **Compliant Code** |
| --- |
| The compliant code block utilizes the template parameter pack instead of the variadic argument. Using this, the add() function safely accepts any number of arguments. The type safety of this method is also secure since it makes use of std::enable\_if to ensure the method only accepts data types of type int. |
| [Compliant code block; code should be indented using 12-point Courier New font.]  #include <type\_traits>  template <typename Arg, typename... Ts, typename std::enable\_if<std::is\_integral<Arg>::value>::type\* = nullptr>  int add(Arg i, Arg j, Ts... all) {  int values[] = { j, all... };  int r = i;  for (auto v : values) {  r += v;  }  return r;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 1: Validate Input – The compliant code ensures input arguments are validated through compile-time checks, like when using std::enable\_if and std::is\_integral, which helps prevent invalid data from being processed.  Principle 4: Keep it simple – Using safer alternatives like parameter packs makes code more maintainable and less error-prone.  Principle 10: Adopt a Secure Coding Standard – Avoiding variadic functions adheres to modern secure coding practices and aligns with the SEI CERT standards. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Function-ellipsis | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL50 | None |
| Clang | 3.9 | Cert-dcl50-cpp | Checked by clang-tidy |
| Helix QAC | 2024.3 | C++20212, C++2625 | None |
| Klockwork | 2024.3 | MISRA.FUNC.VARARG | None |
| LDRA tool suite | 9.7.1 | 41 S | Fully implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL50-a | Functions shall not be defined with a variable number of arguments |
| Polyspace Bug Finder | R2024a | CERT C++: DCL50-CPP | Checks for function definition with ellipsis notation (rule fully covered) |
| Rule Checker | 22.10 | Function-ellipsis | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | FunctionEllipsis | None |

#### Coding Standard 2

| **Coding Standard** | **Label** | Do Not Read Uninitialized Memory |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Uninitialized variables and dynamically allocated memory have indeterminate values, which can result in trap representation, in which attempting to read the value of an object that has trap representation can result in undefined behavior. By implementing this as a security policy, it ensures that memory is properly allocated and prevents an easy avenue for attack.  https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory |

| **Noncompliant Code** |
| --- |
| In this example, the int \* object is allocated by a new expression, however the memory it points to is not initialized. The object’s pointer value and the value it points to are printed to the standard output stream. Attempting to print the value pointed to yields an indeterminate value, thus resulting in undefined behavior an attacker can use for arbitrary code injection. |
| #include <iostream>    void f() {    int \*i = new int;    std::cout << i << ", " << \*i;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, the memory is directly initialized to the value 12 prior to printing the value. This helps in resulting in defined behavior. |
| #include <iostream>    void f() {  int \*i = new int(12);  std::cout << i << ", " << \*i;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 1: Validate Input – Initializing variables ensures all program input is valid and predictable, preventing errors from unpredictable data.  Principle 2: Heed Compiler Warnings – using compiler warnings prevents uninitialized memory access, reducing the likelihood of security vulnerabilities.  Principle 4: Keep it Simple - Explicit initialization simplifies the program behavior, also avoiding unintended side effects from unpredictable data.  Principle 8: Practice Defense in Depth – Incorporating defense-in-depth strategies adds an extra safeguard against vulnerabilities caused by undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local  Uninitialized variable |
| Helix QAC | 2024.3 | DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978 | [Insert text.] |
| Klocwork | 2024.3 | UNINIT.CTOR.MIGHT  UNINIT.CTOR.MUST  UNINIT.HEAP.MIGHT  UNINIT.HEAP.MUST  UNINIT.STACK.ARRAY.MIGHT  UNINIT.STACK.ARRAY.MUST  UNINIT.STACK.ARRAY.PARTIAL.MUST  UNINIT.STACK.MIGHT  UNINIT.STACK.MUST | [Insert text.] |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024a | CERT C++: EXP53-CPP | Checks for:  Non-initialized variable  Non-initialized pointer  Rule partially covered. |
| PVS-Studio | 7.33 | V546, V573, V614, V670, V679, V730, V788, V1007, V1050 |  |
| RuleChecker | 22.10 | uninitialized read | Partially checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | Do Not Attempt to Modify String Literals |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | String literals in C are stored in static, read-only memory and are treated as immutable. Attempting to modify a string literal leads to undefined behavior, which can lead to the possibility of an attack vector. In order to reduce the risk and ensure safety, string literals should always be assigned to const char\* or const wchar\_t\* pointers, which helps keep them immutable.  https://wiki.sei.cmu.edu/confluence/display/c/STR30-C.+Do+not+attempt+to+modify+string+literals |

| **Noncompliant Code** |
| --- |
| In this code example, the char pointer str is initialized to the address of a string literal, thus attempting to modify the string literal results in undefined behavior. |
| char \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal can specify the initial values of characters in an array as well as the size of the array. The code in this example creates a copy of the string literal in the space allocated to the character array str, which can be safely modified. |
| char str[] = "string literal";  str[0] = 'S'; |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**   Principle 1: Validate Input – Ensures string literals are treated as immutable, preventing unintended modifications and enforcing predictable system behavior.  Principle 2: Heed Compiler Warnings – Paying attention to compiler warnings ensures any attempts to modify read-only memory are caught early on, reducing the risks of undefined behavior.  Principle 4: Keep It Simple – Treating string literals as immutable avoids undefined behavior and reduces security risks to potential attacks.  Principle 5: Default Deny – Enforcing immutability of string literals by default ensures safe and secure coding practices through explicit restrictions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | string-literal-modification, write-to-string-literal | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| Compass/ROSE |  |  | Can detect simple violations of this rule |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to "char \*" |
| Helix QAC | 2024.3 | C0556, C0752, C0753, C0754, C++3063, C++3064, C++3605, C++3606, C++3607 | Fully implemented with extensive checks |
| Klocwork | 2024.3 | CERT.STR.ARG.CONST\_TO\_NONCONST, CERT.STR.ASSIGN.CONST\_TO\_NONCONST | Detects unsafe string conversions to ensure compliance |
| LDRA Tool Suite | 9.7.1 | 157 S | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_C\_STR30-a, CERT\_C\_STR30-b | A string literal shall not be modified; Do not modify string literals |
| PC-lint Plus | 1.4 | 489, 1776 | Partially supported |
| Polyspace Bug Finder | R2024a | CERT C: Rule STR30-C | Checks for writing to const qualified object (rule fully covered) |
| PVS-Studio | 7.33 | V675 | Fully implemented, detects unsafe string modifications |
| RuleChecker | 24.04 | string-literal-modification | Partially checked |
| Splint | 3.1.1 |  | No checker specified |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively verified with examples (one compliant and one non-compliant example provided) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP-SQL | SQL injection vulnerabilities occur when user inputs are directly concatenated into SQL queries without proper sanitization or validation. This allows attackers to manipulate SQL queries, potentially exposing or altering sensitive data. To mitigate this, developers must use parameterized queries or prepared statements that separate SQL logic from data inputs, ensuring that user input is treated strictly as data and not as executable code. (*IDS00-J. Prevent SQL Injection - SEI CERT Oracle Coding Standard for Java - Confluence*, n.d.). |

| **Noncompliant Code** |
| --- |
| The java.sql.preparedStatement class properly escapes input strings, which prevents SQL injection when used correctly. This code example modifies the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement. However, the prepared statement still allows a SQL injection attack by using the unsanitized input argument username in the prepared statement. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;    class Login {    public Connection getConnection() throws SQLException {      DriverManager.registerDriver(new              com.microsoft.sqlserver.jdbc.SQLServerDriver());      String dbConnection =        PropertyManager.getProperty("db.connection");      // Can hold some value like      // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"      return DriverManager.getConnection(dbConnection);    }      String hashPassword(char[] password) {      // Create hash of password    }      public void doPrivilegedAction(      String username, char[] password    ) throws SQLException {      Connection connection = getConnection();      if (connection == null) {        // Handle error      }      try {        String pwd = hashPassword(password);        String sqlString = "select \* from db\_user where username=" +          username + " and password =" + pwd;        PreparedStatement stmt = connection.prepareStatement(sqlString);          ResultSet rs = stmt.executeQuery();        if (!rs.next()) {          throw new SecurityException("User name or password incorrect");        }          // Authenticated; proceed      } finally {        try {          connection.close();        } catch (SQLException x) {          // Forward to handler        }      }    }  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the solution uses a parameterized query with a ? character as a placeholder for the argument. It also validates the length of the username argument, which helps prevent an attacker from inputting an arbitrarily long username. |
| public void doPrivilegedAction(    String username, char[] password  ) throws SQLException {    Connection connection = getConnection();    if (connection == null) {      // Handle error    }    try {      String pwd = hashPassword(password);        // Validate username length      if (username.length() > 8) {        // Handle error      }        String sqlString =        "select \* from db\_user where username=? and password=?";      PreparedStatement stmt = connection.prepareStatement(sqlString);      stmt.setString(1, username);      stmt.setString(2, pwd);      ResultSet rs = stmt.executeQuery();      if (!rs.next()) {        throw new SecurityException("User name or password incorrect");      }        // Authenticated; proceed    } finally {      try {        connection.close();      } catch (SQLException x) {        // Forward to handler      }    }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 1: Validate Input – Checking and sanitizing all input before it goes into SQL queries ensures that only safe data is processed, preventing malicious injections.  Principle 3: Architect and Design for Security Policies – Designing the system to use prepared statements and parameterized queries builds security into the structure, reducing SQL injection risks.  Principle 6: Adhere to the Principle of Least Privilege – Restricting account access to only the permissions they need limits the potential for an attack or exploit.  Principle 8: Practice Defense in Depth – Combining strategies like input validation, parameterized queries, and secure database configurations ensures multiple layers of defense against attacks.  Principle 9: Use Effective Quality Assurance Techniques – Techniques like code reviews, testing, and audits during development help catch potential SQL injection and other errors early, thereby reducing the potential for attacks later on. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| CodeSonar | 8.1p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Coverity | 7.5 | SQLI, FB.SQL\_PREPARED\_STATEMENT\_GENERATED, FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Fortify | 1.0 | HTTP\_Response\_Splitting, SQL\_Injection\_Persistence, SQL\_Injection | Implemented |
| Klocwork | 2024.3 | SV.DATA.DB, SV.SQL, SV.SQL.DBSOURCE | Implemented |
| Parasoft Jtest | 2024.1 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SonarQube | 9.9 | S2077, S3649 | Executing SQL queries is security-sensitive |
| SpotBugs | 4.6.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE, SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Properly Deallocate Dynamically Allocated Resources** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Dynamically allocated memory must be properly deallocated to avoid memory leaks. Such functions exist to perform correct deallocation like delete for new and delete[] for new[], each of which ensure that memory is returned to the system safely. Improper use of memory can result in undefined behavior, and thus security vulnerabilities.  https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources |

| **Noncompliant Code** |
| --- |
| In this code example, the local variable space is used as an argument to the new operator. The resulting pointer is then passed to ::operator delete(), leading to undefined behavior because ::operator delete() attempts to free memory that was not originally allocated by ::operator new() |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;    // ...    delete s1;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, the call to ::operator delete() is removed and is instead explicitly calling s1’s destructor. This particular situation is warranted for explicitly invoking a destructor. |
| #include <iostream>    struct S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {    alignas(struct S) char space[sizeof(struct S)];    S \*s1 = new (&space) S;      // ...      s1->~S(); |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 4: Keep it Simple – Ensuring that dynamically allocated memory is properly deallocated helps avoid unintended side effects like memory leaks or undefined behavior, ensuring predictable and stable system performance.  Principle 6: Adhere to the Principle of Least Privilege – By ensuring that memory is effectively managed will help reduce unnecessary use of resources, thereby minimizing opportunities for misuse.  Principle 8: Practice Defense in Depth – Properly deallocation memory acts as an additional layer of security against vulnerabilities.  Principle 9: Use Effective Quality Assurance Techniques – Memory testing ensures vulnerabilities related to improper usage are identified and resolved as early as possible. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | invalid\_dynamic\_memory\_allocation dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.20 | CertC++-MEM51 |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 8.1p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Free non-heap variable  Double free  Type mismatch  Leak |
| Helix QAC | 2024.3 | CL.FFM.COPY  CL.FMM  CL.SHALLOW.ASSIGN  CL.SHALLOW.COPY  FMM.MIGHT  FMM.MUST  FNH.MUST  FUM.GEN.MIGHT  UNINIT.CTOR.MIGHT  UNINIT.CTOR.MUST  UNINIT.HEAP.MIGHT  UNINIT.HEAP.MUST |  |
| Klocwork | 2024.3 | CERT.STR.ARG.CONST\_TO\_NONCONST  CERT.STR.ASSIGN.CONST\_TO\_NONCONST |  |
| LDRA tool suite | 9.7.1 | 232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 475 S, 483 S, 484 S, 485 S, 64 D, 112 D | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free  Always provide empty brackets [] for delete when deallocating arrays  Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor  Properly deallocate dynamically allocated resources |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024a | CERT C++: MEM51-CPP | Checks for:  - Invalid deletion of pointer  - Invalid free of pointer  - Deallocation of previously deallocated pointer  Rule partially covered |
| PVS-Studio | 7.33 | V515, V554, V611, V701, V748, V773, V1066 |  |
| SonarQube C/C++ Plugin | 4.10 | S1232 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Static Assertions to Test Constant Expressions** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are a diagnostic tool meant for identifying and eliminating software defects that may result in vulnerabilities. It is better to use static assertions to test constant expressions instead of the runtime assert() macro, due to the limitations with the macro incurring a runtime overhead and subsequently calling abort(). It is more useful in identifying suspected incorrect assumptions instead of runtime checking.  https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression |

| **Noncompliant Code** |
| --- |
| This code example uses the assert() macro to assert a memory-mapped structure essential for the code to behave as intended. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| The compliant code uses conditional checking instead of using the assert function at runtime. |
| # struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif    s1->~S();} |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 1: Validate Input – Verifying assumptions about the size of structs before the program runs ensures that they align with coding standard expectations, thereby reducing any potential runtime errors.  Principle 2: Heed Compiler Warnings – Using compile-time checks like static assertions helps identify and fix potential issues during development, avoiding the possibility for introducing vulnerabilities during deployment  Principle 4: Keep It Simple – Simplifying the debugging process by catching issues early with compile-time checks makes the code easier to understand and maintain, also making it easier to identify any potential bugs or errors as early as possible. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 | [Insert text.] |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 8.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle All Exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exceptions that are uncaught can lead to undefined termination. When an exception is caught the program terminates using std::terminate(), which may allow various resources to be accessed. While catching exceptions may not be handled gracefully, a controlled catch of an exception helps prevent security vulnerabilities.  https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither the f() nor main() catch exceptions thrown by the throwing\_func() method. Since no matching handler is found, std::terminate() is called, leading to an undefined termination. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, ensuring the stack is unwound up to the main() function and allows for a controlled exception. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 2: Heed Compiler Warnings – Handling exceptions correctly prevents unhandled exceptions from causing compiler warnings, allowing the program to pass static check analysis.  Principle 4: Keep It Simple – Explicit exception handling invites better error management, making the program easier to debug and thereby reducing the risk of undefined behaviors.  Principle 8: Practice Defense in Depth – Robust exception handling adds an additional layer of protection, reducing the risk of failures caused by unhandled exceptions.  Principle 9: Use Effective Quality Assurance Techniques – Rigorous exception handling testing ensures vulnerabilities are identified and fixed, thus improving the program's overall security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP.ERR51-a, CERT\_CPP.ERR51-b | Always catch exceptions. Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point. |
| Polyspace Bug Finder | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered). |
| Astrée | 22.10 | main-function-catch-all, early-catch-all | Partially checked. |
| RuleChecker | 22.10 | main-function-catch-all, early-catch-all | Partially checked. |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented. |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| Helix QAC | 2024.3 | C++4035, C++4036, C++4037 |  |
| Klocwork | 2024.3 | MISRA.CATCH.ALL |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Use correct integer precisions** |
| --- | --- | --- |
| Integers | INT35-008-CPP | It’s imperative as a coding standard to use correct integer precisions. The incorrect use of integer precision can result in undefined behavior due to the system incorrectly assuming the numeric range of integer types. Padding bits may be included to inflate the size of an integer type without increasing its precision. This standard serves to ensure that correct precision is accounted for, thus avoiding vulnerabilities caused by incorrect assumptions.  https://wiki.sei.cmu.edu/confluence/display/c/INT36-C.+Converting+a+pointer+to+integer+or+integer+to+pointer |

| **Noncompliant Code** |
| --- |
| The pow2() function limits shifts based on the size of unsigned int, but it fails on systems with padding bits, causing undefined behavior. |
| #include <limits.h>    unsigned int pow2(unsigned int exp) {  if (exp >= sizeof(unsigned int) \* CHAR\_BIT) {  /\* Handle error \*/  }  return 1 << exp;  } |

| **Compliant Code** |
| --- |
| The popcount() function accurately calculates integer precision by counting set bits, ensuring portability and preventing undefined behavior. |
| #include <stddef.h>  #include <stdint.h>    /\* Returns the number of set bits \*/  size\_t popcount(uintmax\_t num) {  size\_t precision = 0;  while (num != 0) {  if (num % 2 == 1) {  precision++;  }  num >>= 1;  }  return precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 1: Validate Input – Ensuring integer types stay within range makes the program more reliable and prevents unintended behavior.  Principle 2: Heed Compiler Warnings – Addressing warnings related to shifts and padding bits eliminates potential sources of bugs or system incompatibilities.  Principle 4: Keep It Simple – Using functions like popcount() makes the code easier to understand and minimizes the risk of errors.  Principle 8: Practice Defense in Depth – Handling integer precision correctly adds an extra safety step, protecting the program from errors from unexpected system behavior.  Principle 10: Adopt a Secure Coding Standard – Using clear methods like popcount() ensures the program is written securely and consistently across different platforms. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | LANG.ARITH.BIGSHIFT | Reports overflows due to insufficient precision. |
| CodeSonar | 8.1p0 | LANG.ARITH.BIGSHIFT | Shift Amount Exceeds Bit Width. |
| Cppcheck Premium | 24.9.0 | premium-cert-int35-c | Fully implemented. |
| Helix QAC | 2024.3 | C0582, C++3115 | Use correct integer precisions when checking the right-hand operand of the shift operator. |
| Parasoft C/C++test | 2023.1 | CERT\_C-INT35-a | Checks for correct integer precisions during bit shifts. |
| Polyspace Bug Finder | R2024a | CERT C: Rule INT35-C | Checks for situations when integer precisions are exceeded (fully covered). |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not modify objects with temporary lifetime** |
| --- | --- | --- |
| Expressions | EXP35-009-CPP | Modifying objects with temporary lifetime results in undefined behavior. This standard enforces proper handling of temporary objects, ensuring code conforms to the current C standard, which extends temporary object lifetimes compared to older versions. Developers adhering to this coding standard can prevent undefined behavior, thereby reducing the possibility of an attack avenue.  https://wiki.sei.cmu.edu/confluence/display/c/EXP35-C.+Do+not+modify+objects+with+temporary+lifetime |

| **Noncompliant Code** |
| --- |
| This code is noncompliant despite adhering to the C11 standard due to failing to confirm to the C99 standard, it results in undefined behavior because the sequence point before the call to printf() comes between the call and access of printf() of the string in the returned object. |
| #include <stdio.h>    struct X { char a[8]; };    struct X salutation(void) {  struct X result = { "Hello" };  return result;  }    struct X addressee(void) {  struct X result = { "world" };  return result;  }    int main(void) {  printf("%s, %s!\n", salutation().a, addressee().a);  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that a pre-C11 compiler will fail to compile, rather than risk undefined behavior. |
| #include <stdio.h>    #if \_\_STDC\_VERSION\_\_ < 201112L  #error This code requires a compiler supporting the C11 standard or newer  #endif    struct X { char a[8]; };    struct X salutation(void) {  struct X result = { "Hello" };  return result;  }    struct X addressee(void) {  struct X result = { "world" };  return result;  }    int main(void) {  printf("%s, %s!\n", salutation().a, addressee().a);  return 0;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 1: Validate Input – Checks the inputs and results from functions and that they meet the rules, thus reducing the chance for errors or undefined behavior.  Principle 2: Heed Compiler Warnings – The C11 compiler helps developers catch and fix problems early on by providing helpful warnings and errors, making it easier to identify what needs to be corrected before compiling.  Principle 4: Keep it Simple – Defining how temporary objects are used makes the code easier to understand and reduces the risk of easy mistakes.  Principle 8: Practice Defense in Depth – Implementing other compiler standards acts as an additional layer of defense, thus ensuring that the program avoids issues related to temporary objects. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | temporary-object-modification | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-EXP35 | Partially implemented |
| CodeSonar | 8.1p0 | LANG.CAST.ARRAY.TEMP | Array to Pointer Conversion on Temporary Object |
| Cppcheck Premium | 24.9.0 | premium-cert-exp35-c | Fully implemented |
| Helix QAC | 2024.3 | C0450, C0455, C0459, C0464, C0465, C++3807, C++3808 | Enhanced Enforcement |
| LDRA tool suite | 9.7.1 | 642 S, 42 D, 77 D | Do not modify objects with temporary lifetime |
| Parasoft C/C++test | 2023.1 | CERT\_C\_EXP35-a | Checks for accesses on objects with temporary lifetime (rule fully covered) |
| Polyspace Bug Finder | R2024a | CERT-C: Rule EXP35-C | Checks for accesses on objects with temporary lifetime (rule fully covered) |
| Splint | 3.1.1 |  | Not Specified |
| RuleChecker | 24.04 | temporary-object-modification | Partially checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| Memory Management | STD50-010-CPP | Accessing memory through pointers that reference deallocated memory is undefined behavior and risks vulnerabilities. This standard ensures dynamically allocated memory is not accessed after it has been freed, thus preventing undefined behavior and potential security risks.  https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory |

| **Noncompliant Code** |
| --- |
| The code deallocates a pointer using delete and then dereferences it to call a member function, leading to undefined behavior. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f(); // Accessing deallocated memory  } |

| **Compliant Code** |
| --- |
| The compliant code ensures the dynamically allocated memory is used before deallocation, avoiding the risk of accessing a dangling pointer. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f(); // Safely using the memory  delete s; // Deallocate after use} |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle 2: Heed Compiler Warnings – Using modern compilers helps identify potential issues with dangling pointers, making the code safer by fixing problems early.  Principle 4: Keep It Simple – Organizing memory allocation and cleanup in a clear and logical way avoids confusion and prevents errors like accessing freed memory.  Principle 8: Practice Defense in Depth – Preventing dangling pointers adds an extra safety layer, reducing the chances of memory issues causing unexpected behavior.  Principle 9: Use Effective Quality Assurance Techniques – Using tools like static analysis, runtime checks, and fuzz testing helps catch dangling pointer problems, ensuring the program runs reliably. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use | Detects use of dangling pointers. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 | Identifies violations of pointer safety guidelines. |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 8.1p0 | ALLOC.UAF | Detects use after free scenarios. |
| Compass/ROSE | v7.5.0 | USE\_AFTER\_FREE | Detects specific instances where memory is deallocated more than once or read/written to the target of a freed pointer. |
| Helix QAC | 2024.3 | C++4303, C++4304 | Ensures memory safety when pointers are dereferenced after deallocation. |
| Klocwork | 2024.3 | UFM.DEREF.MIGHT  UFM.DEREF.MUST  UFM.FFM.MIGHT  UFM.FFM.MUST  UFM.RETURN.MIGHT  UFM.RETURN.MUST  UFM.USE.MIGHT  UFM.USE.MUST | Monitors use of dangling pointers and ensures safe pointer dereferencing. |
| LDRA Tool Suite | 9.7.1 | 483 S, 484 S | Partially implemented checks for use of freed memory. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM50-a | Identifies and prevents usage of resources after they are freed. |
| Parasoft Insure++ |  |  | Provides runtime detection for memory violations. |
| Polyspace Bug Finder | R2024a | CERT C++: MEM50-CPP | Checks for pointer access out of bounds, deallocation of previously deallocated pointers, and use of previously freed pointers. |
| PVS-Studio | 7.33 | V586, V774 | Detects memory deallocation and use-after-free violations. |
| Splint | 5.0 |  | Provides partially implemented checks for dangling pointer use. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Integrating security measures into each phase of the DevOps lifecycle, the DevSecOps diagram acts as an outline with how Green Pace can effectively enforce compliance with the security policy. During the Assess and Plan phase, tools that provide threat modeling and risk analysis help align with the diagram’s focus on being proactive with threat detection. The Design and Build phases each incorporate IDE plugins and automated security testing that maps directly to the diagram’s emphasis on integrating security into development workflows.

As code moves to the Verify and Test phase, dynamic and static application security testing (DAST/SAST) and continuous vulnerability scans reflect the diagram’s focus on automated validation and quality assurance. The Transition, Monitor and Respond phases can utilize various tools for automation and compliance; for instance, tools like Splunk can be used for monitoring, OpenSCAP can be used for compliance checks, and SOAR Splunk Phantom can be used for automated incident response (DeVito, 2024). Together these help Green Pace align with the diagram’s objectives in maintaining security with threat detection. Lastly, by leveraging the DevSecOps diagram and integrating these tools into its infrastructure, Green Pace can achieve automated, robust security compliance.

DeVito, A. (2024, May 10). 25 Top DevSecOps Tools (Ultimate Guide for 2024). *StationX*. <https://www.stationx.net/top-devsecops-tools/>

GeeksforGeeks. (2024h, July 8). *What is SOAR (Security Orchestration, Automation and Response) ?* GeeksforGeeks. <https://www.geeksforgeeks.org/what-is-soar/?ref=header_outind>

Vaghela, E. (2024, November 13). *How Automation is Transforming DevSecOps for Better Security*. DEV Community. <https://dev.to/enna/how-automation-is-transforming-devsecops-for-better-security-1co>

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Probable | Medium | P12 | L1 |
| STD-002-CPP | High | Probable | Medium | P12 | L1 |
| STD-003-CPP | Low | Likely | Low | P9 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CPP | Low | Probable | Medium | P4 | L3 |
| STD-010-CPP | High | Likely | Medium | P18 | L1 |

### Create Policies for Encryption and Triple A

Include all three types of encryptions (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest affirms that data that is stored on various storage devices is protected from unauthorized access. Protection data through encryption methods ensures that even if a device or account is compromised, the data that is stored will still be safe. This policy serves to implement protection on various devices and applications like hardware devices, database systems, or file storage technologies. Common encryption standards like the Advanced Encryption Standard-256 are often employed that offer robust encryption protection. |
| Encryption in flight | Encryption in flight pertains to data that is being transferred between various systems or networks. Implementing strong encryption protocols like Transport Layer Security (TLS) can ensure that data being transferred remains secure against potential attacks or eavesdropping attempts. This security policy aims to keep unauthorized parties from accessing data while it is being transmitted between various applications or systems, either through common API endpoints, servers, and other applications that utilize cross communication methods. |
| Encryption in use | Encryption in use protects data that is actively being accessed, modified, or processed by an application. Implementing a security policy for data in use ensures that sensitive information remains secure throughout its usage by individuals or applications. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user seeking to be granted access to a system or network. It is used in validating the credentials of legitimate users, so they can access the necessary resources required. Authentication includes the use of various sign-on methodologies, such as passwords, biometrics, single sign-on (SSO’s), and digital certificates. “Authentication is the first step in the AAA security process and describes the network or application’s way of identifying a user and ensuring the user is whom they claim to be (Mylonas, 2018)”. This policy seeks to embrace authentication in helping keep systems protected from unauthorized access. |
| Authorization | Authorization determines the resources and privileges granted to individuals to access. Typically, this includes setting policies that restrict certain actions for users, depending on the user’s role, or permission for access. Users are typically provided with certain levels of access, ensuring they only have access based on the principle of least privilege, meaning a user should only have access to what the minimum requirement is required to perform their specific job function. “Users are assigned authorization levels that define their access to a network and associated resources (Mylonas, 2018).” |
| Accounting | Accounting pertains to monitoring and logging user activities, specifically, the time they accessed something, what they accessed, and any commands they executed. This helps ensure that certain insights are accounted for, especially during security analysis or auditing. This process helps administrators identify any potential security incidents and isolate a user’s specific steps taken. As mentioned, “Proper accounting enables network and system administrators to review who has been attempting to access what and if access was granted (Mylonas, 2018).” This security policy illustrates the effectiveness of implementing accounting in preventing potential attacks or incidents based on user activity. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by the chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 11/17/2024 | Module Three Milestone | Dylan Cavazos |  |
| 1.2 | 12/08/2024 | Module Six Project One | Dylan Cavazos |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |